Rapport de projet collectif

Jouer au morpion avec un robot
Projet N°13 – Tuteur : N. RUTY

ORCESI Astrid
PRABLANC Pierre
ROCHE Fanny
SIRINELLI Olivier

Année Universitaire : 2013 - 2014

REMERCIEMENTS

Nous remercions tout particulièrement M. RUTY, notre tuteur, qui a su nous conseiller et nous diriger dans l'élaboration du projet. Ce projet n'aurait sans doute pas été le même sa formidable imprimante 3D qui a imprimé nos magnifiques pions.

Nous tenons également à remercier M. FRISTOT et M.VOISIN-FRADIN pour leur encadrement tout au long de la réalisation du projet.

Il aurait été bien difficile de monter le robot, de l'alimenter et donc de le faire fonctionner sans l'aide précieuse de tout le personnel présent au 3e étage de Minatec, un grand merci à eux pour leur patience face à nos nombreuses sollicitations.

Enfin, nos sincères remerciements vont au groupe de 3e année SMPB qui nous a précédés dans l'élaboration de ce projet.

SOMMAIRE

Introduction

De nos jours, la robotique occupe une place de plus en plus importante dans l'industrie. Des bras dans les chaînes de montage aux robots équipés pour la chirurgie, des notions en asservissement et pilotage deviennent alors indispensables pour les ingénieurs.

Plus modeste, notre projet consistait en la conception d'un système permettant de jouer au morpion en utilisant un bras robotique piloté par une carte compatible Arduino. L'idée était de contrôler le bras robot via un ordinateur communiquant avec la carte à l'aide d'une interface graphique simple et intuitive. L'objectif final était de permettre le pilotage du robot sans contact, utilisant un capteur de réalité augmentée, le Leap Motion. A terme, la combinaison des deux technologies pourra permettre une utilisation plus performante des outils de robotique (bras chirurgicaux commandés par les mains du chirurgien à l'autre bout du monde …).

Ce rapport détaille l'ensemble de nos travaux sur ce sujet ainsi que les résultats obtenus. Il ne saurait cependant être complet sans une démonstration : Une série de vidéos peuvent se trouver à l'adresse suivante : http://sirinelli.fr/granola/

Avertissement : Le travail présenté ci-dessous a permis l'élaboration d'un prototype. Le code fourni en annexe, bien que fonctionnel, n'est en aucun cas optimisé.

Remarque : Ce rapport apporte de nombreux éléments pour continuer ou améliorer le projet. Il est possible que certains éléments manquent, auquel cas ne pas hésiter à nous contacter.

I. Cahier des charges

Nous ont été fournis au début du projet un bras robot à 6 degrés de liberté *DFRobot ROB0036*, une carte *DFRduino Romeo V1.1* ainsi qu'un Leap Motion. Les explications relatives à ces différents composants sont fournies dans ce rapport. Pour les datasheets et manuels, se référer aux annexes et à la bibliographie.

1. Les objectifs du projet

Le projet se décompose en une suite de quatre objectifs distincts :

- Tout d'abord il est nécessaire de savoir contrôler le bras robot à l'aide d'un programme développé sous IDE Arduino.
- Le second objectif de mi projet est de pouvoir diriger le bras robot à partir d'une interface graphique implémentée sur l'ordinateur.
- Nous avons à notre disposition un capteur infra rouge, le Leap Motion, capable de détecter le bout des doigts présents au-dessus de sa surface. Parallèlement à l'objectif précédent, il faut donc développer un programme capable de récupérer les informations utiles du Leap Motion pour contrôler le bras robot pas à pas.
- Enfin, l'objectif final du projet est de relier le Leap Motion au bras robot, selon le synopsis suivant, dans le but de jouer au à partir des ordres donnés par le mouvement des mains du joueur.

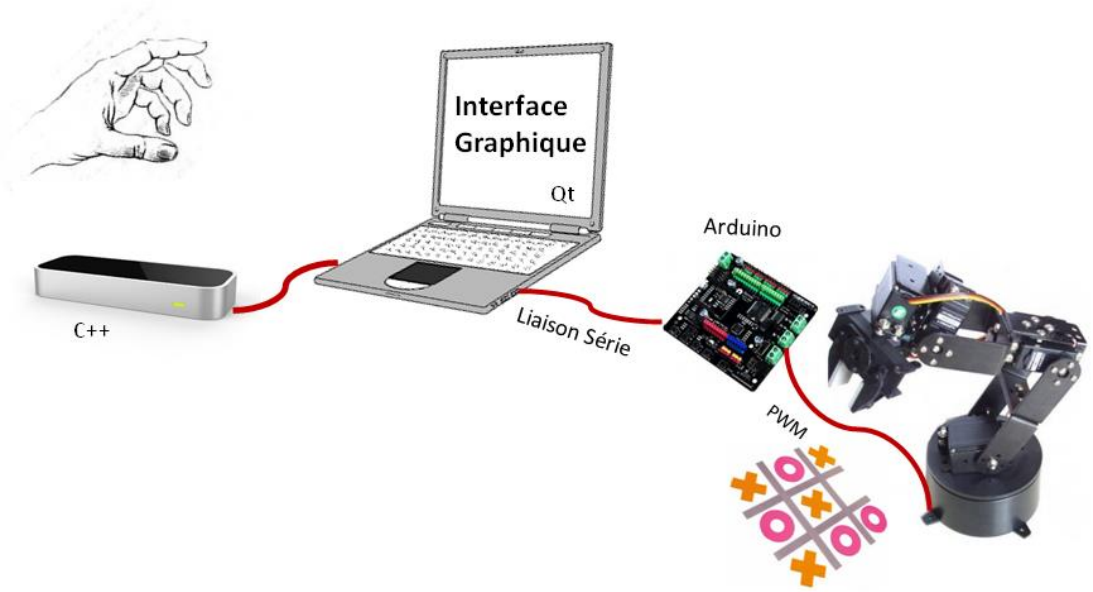


Figure 1 : Schéma de principe du système voulu

2. Choix techniques

a. Section Bras robot

i. La carte et le bras robot

Couplage de la carte Romeo et du robot :

La carte Romeo est une carte 'Tout-en-un' spécialement conçue pour la robotique. Elle bénéficie de la plateforme de développement Arduino. Les cartes de développement Arduino embarquent un microcontrôleur ainsi que des périphériques d'entrées/sorties. Arduino fournit des librairies codées en C permettant d'exploiter facilement les périphériques de la carte. Précisément, nous utiliserons les 6 sorties PWM pour piloter les servomoteurs ainsi que la liaison série pour communiquer avec l'ordinateur. La carte est alimentée par le 5V de la liaison série, mais les servomoteurs consomment trop de courant. Il est donc nécessaire de connecter une source d'alimentation externe sur le bornier Servo Power Input. Nous utilisons donc une alimentation continue INSTEK (modèle GPS-3030DD) réglé sur 7V, avec une limitation de 1.90A sachant que les servomoteurs sont généralement alimentés entre 5V et 7.2V.

Sortie de la carte Romeo :

Comme nous venons de l'expliquer ci-dessus, la carte Romeo envoie au robot les sorties sous forme de génération de PWM (Pulse-Width Modulation). Cela fonctionne de la manière suivante : la carte génère un signal rectangulaire à une fréquence de 50 Hz environ dont l'information concernant l'ordre envoyé au servomoteur réside dans le rapport cyclique. En effet, chaque valeur de ce rapport cyclique correspond à un angle bien défini. Typiquement on a :

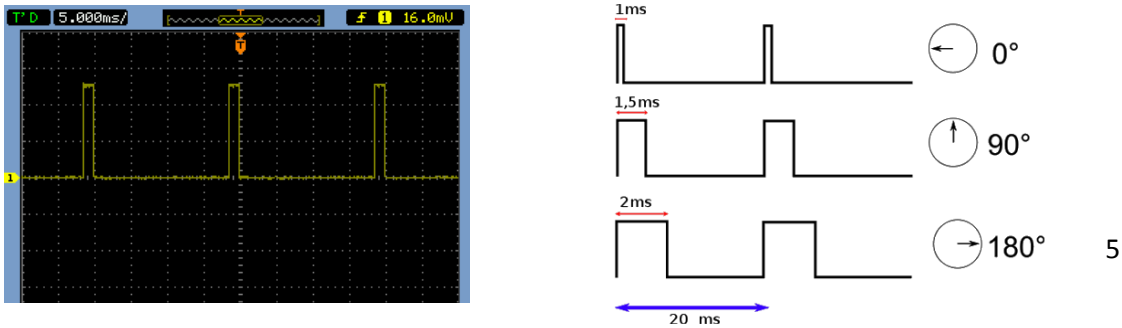


Figure 2 : oscillogramme et schéma d'une PWM

En ce qui concerne notre projet, on a pu observer la PWM envoyée par exemple au servomoteur contrôlant la pince du bras de robot à l'oscilloscope. On observe alors les oscillogrammes suivant :

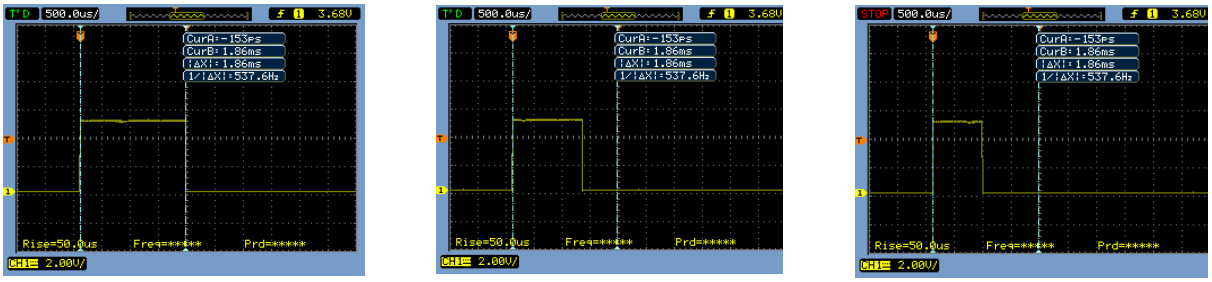


Figure 3 : oscillogrammes de la PWM pour respectivement la pince fermée, la pince semie ouverte (en possession d'un pion) et ouverte

ii. L'interfaçage

Pour des soucis de portabilité nous avons choisi le langage C++ pour réaliser nos programmes. En effet, l'autre possibilité était d'utiliser Matlab mais ce logiciel, bien que portable, est assez cher et par conséquent pas accessible à tous.

Successeur du célèbre langage de programmation C, le C++ orienté objet, forts de ces nombreuses librairies, dispose d'une myriade de fonctions.

Pour compléter ceci, on va utiliser l'interface de programmation (API en Anglais) Qt qui fournit de nombreux éléments pour entre autre concevoir des interfaces graphiques.

Loin de l'exploiter à son maximum, nous choisissons d'utiliser Qt pour la nôtre et pour dialoguer avec le robot via la liaison série. Son grand avantage est une portabilité vers tous les systèmes d'exploitation ce qui fait partie de nos objectifs. De plus, Qt dispose d'une grande variété de Widgets (boutons, sliders…) permettant de faire des interfaces graphiques très riches.

Pour programmer rapidement avec Qt, on utilise l'IDE *QtCreator* présentant l'immense intérêt de gérer la compilation par lui-même, en plus des facilités classiques d'un IDE.

L'objectif de l'interface graphique est de mettre en place l'espace de jeu sur une fenêtre pour permettre au joueur d'une part de contrôler le robot en cliquant sur les boutons de la fenêtre et d'autre part de savoir à quel stade il en est de la partie. De plus, par soucis de précision et également de pouvoir contrôler le robot sans forcément lui donner une trajectoire correspondant aux positions de jeu, nous représenterons donc sur cette interface un moyen permettant de contrôler les servomoteurs indépendamment les uns des autres.

b. Section Leap Motion

Le choix d'un capteur permettant d'obtenir des informations sur la position de la main et des doigts dans l'espace s'est imposé par lui-même. En effet, l'école était déjà en possession d'un capteur qui intègre des traitements permettant d'accéder directement aux informations de position citées précédemment. Compte tenu des contraintes de temps, la réalisation de ce capteur n'aurait pas été possible, ce qui justifie l'utilisation d'un élément existant déjà dans l'industrie. Ce capteur nommé Leap Motion est tout à fait adapté à notre application car il permet de relever avec une grande précision la position des doigts et de la main, à la différence d'un capteur Kinect qui convient davantage aux mouvements du corps.

Le Leap Motion est un système propriétaire que l'on peut connecter par USB 3.0. Il n'est pas possible de connaitre précisément son fonctionnement interne car le hardware et le software sont protégés.

Néanmoins, on parvient à trouver quelques informations sur le fonctionnement de celui-ci. L'appareil est composé de 2 capteurs CCD tous deux indispensables pour obtenir l'information sur la profondeur. En d'autres termes, si on masque l'un des capteurs, on perd les données sur un axe.
Le système intègre également 3 DEL IR qui servent à bombarder la zone d'intérêt afin de récupérer les rayons issus de la réflexion contre la main.

Les caractéristiques annoncées par le constructeur sont :
Détection : Jusqu'à 10 doigts
Degrés de liberté : 6 (3 Dimensions spatiales + 3 Orientations spatiales)
Précision de la position : 1/100 de mm
Fréquence : 200Hz (précision dans le suivi des mouvements)
Zone d'interaction : 0.227 m^3 répartie comme la figure suivante :

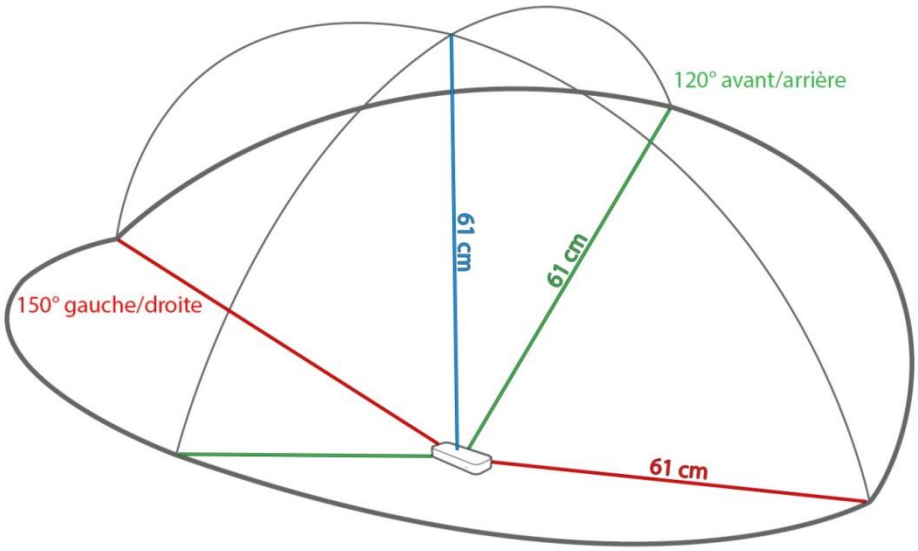


Figure 4 : Champ opérationnel de Leap Motion

D’un point de vue du software, le site du Leap Motion donne les différents langages dans lesquels il est possible de développer une application : C++, C#, Objective-C, Java, JavaScript, Python. Le choix du langage s’est donc porté sur le C++ afin de conserver le même langage de programmation que pour le bras robot. L’équipe travaillant sur le Leap Motion évoluait sur Windows. Le choix de l’environnement de développement s’est alors porté sur Microsoft Visual C++ Express qui a l’avantage d’être gratuit.

II. Conception

1. Bras robot

a. Communication

La communication avec la carte se fait via une liaison série USB. Celle-ci, en théorie, l'alimente également, mais pour assurer un fonctionnement optimal des servomoteurs, une entrée permettant d'alimenter ces derniers est prévue sur la carte.

Pour faire communiquer notre programme Qt avec la liaison série et donc la carte, on utilisera la librairie *QextSerialPort* fournissant les méthodes permettant d'initialiser la connexion et de transférer des données. Pour son installation, on se référera au site de la librairie, donné en annexe dans le manuel du programmeur.

Voici un code typique permettant d'initialiser une liaison série par Qt :

```
m_serialPort = new QextSerialPort("/dev/ttyACM0"); // établit
la connexion avec le port

if     (     !m_serialPort->open(     QIODevice::ReadWrite     |
QIODevice::Unbuffered ) )
{
    std::cout << "Le Robot n'est pas là";
    exit(EXIT_FAILURE);
}
else
{
    m_serialPort->setBaudRate(BAUD9600);
    m_serialPort->setFlowControl(FLOW_OFF);
    m_serialPort->setParity(PAR_NONE);
    m_serialPort->setDataBits(DATA_8);
    m_serialPort->setStopBits(STOP_1);
}
```

Remarque 1 : On notera que ce mode minimaliste ne permet pas la gestion d'un autre port série que celui passé en argument du constructeur (ici **/dev/ttyACM0** pour Unix).

Remarque 2 : Pour le cas de Windows, les ports série ont pour nom COM**N**. Pour connaître le port auquel la carte est connectée, il suffit d'aller dans le panneau de configuration > gestion de périphérique et de trouver la carte dans la liste.

Une méthode dans l'interface graphique sera utilisée pour envoyer les données au robot utilisant la liaison série nouvellement créée. Pour l'appeler on tirera parti des *signals* et *slots* qu'offrent Qt.

b. Interface graphique

En ce qui concerne la conception de notre interface graphique, nous avons donc fait le choix, pour répondre au cahier des charges que nous avons explicité précédemment, de la présenter sous la forme d'une fenêtre possédant plusieurs boutons : 9 représentant les différentes cases du plateau de jeu de morpion, 2 autres représentant les deux équipes qui jouent l'une contre l'autre, ainsi qu'un bouton permettant de commencer une nouvelle partie et ainsi de réinitialiser le jeu.

De plus, afin de contrôler le bras de robot pour des mouvements qui ne font pas forcément partie du « mode de jeu » en tant que tel et conserver une autonomie de déplacement du robot, nous avons décidé de mettre en place 6 *sliders* reliés chacun à un servomoteur du bras. De plus, ces *sliders* sont reliés à des afficheurs LCD permettant d'afficher la valeur de l'angle auquel se trouve chaque servomoteur.

Finalement, afin de pouvoir se remettre en position de jeu après n'importe quel mouvement, nous avons créé un bouton « remise en position de repos » qui permet au robot, comme cela est indiqué dans le texte du bouton, de remettre le robot en position de repos.

Nous avions également pour projet de mettre en place un bouton permettant la réinitialisation des coordonnées correspondant aux cases de jeu etc, mais par manque de temps, nous n'avons pas pu mener cela à bien.

Tout ce qui vient d'être énuméré a été développé en C++ sous l'IDE Qt comme spécifié dans le cahier des charges.

2. Leap Motion

a. Choix de conceptions des mouvements interprétés

L'objectif initial du module construit autour du Leap Motion était d'interpréter les mouvements des mains afin que le robot bouge en conséquence. De nombreuses possibilités s'offraient à nous.

D'abord nous avons pensé à traduire un mouvement de translation venant du Leap Motion en une combinaison de rotations pour le robot. Les équations permettant de formaliser ce problème sont des équations de cinématique inverse. Ces équations ont une formulation différente selon le nombre de degrés de liberté du robot. De plus, l'établissement de ces équations n'est pas chose aisée car ce sujet est encore un domaine de recherche. Dans l'hypothèse où ces équations de cinématique inverse avaient été établies, des problèmes de temps de calculs auraient pu intervenir.

Nous avons donc préféré nous pencher vers deux possibilités plus simples, plus facilement implémentables et donc avec une probabilité de réussite plus grande. La première alternative consistait à commander chaque servomoteur individuellement à l'aide de 2 mains. Le système étant conçu pour droitier, la main gauche appellerait le numéro du moteur et la main droite en contrôlerait la déviation angulaire par un mouvement de translation verticale. La deuxième possibilité envisagée était plus spécifique au jeu du morpion. Le principe était de créer un damier virtuel sur lequel on pourrait sélectionner les cases du jeu en les « pinçant ». Le robot n'aurait alors plus qu'à interpréter cet ordre venant du Leap Motion.

Finalement, ce sont les deux dernières possibilités qui ont été retenues car elles tenaient dans les contraintes de temps et de faisabilité.

b. Organisation du code

Pour développer notre code, nous utilisons les librairies du kit de développement ou SDK (Software Development Kit) fournies par le site du Leap Motion. Ces librairies contiennent les

éléments de base permettant récupérer les informations provenant du capteur. Ce code est en grande partie protégé et donc impossible à analyser. Néanmoins, il est suffisamment commenté pour en connaitre l'utilisation. Le SDK du Leap Motion (ou Leap SDK) contient les solutions générés pour Visual C++ pour les versions 2008, 2010 et 2012.
Toutes les classes que nous utilisons sont prototypées dans la librairie leap.h. Les classes nécessaires au fonctionnement de base de notre code sont les suivantes :

```
class Controller : public Interface
class Listener
```

La classe `Controller` hérite de la classe `Interface` qui est de type assez obscur (LEAP_EXPORT_CLASS). Elle permet de créer un objet permettant de récupérer les trames de données du Leap Motion. Cependant, les données ne peuvent être utilisées qu'à travers la méthode `Listener`. Toutes les méthodes de la classe `Controller` prennent en argument une référence sur la classe `Listener`. Il y a plusieurs façons de créer un objet `Controller`. On peut utiliser soit :

- le constructeur `Controller(Listener& listener)` qui permet d'instancier l'objet `Controller`.
- Le constructeur par défaut `Controller()` suivi de la méthode `bool addListener(Listener& listener)` qui permet d'ajouter un `listener` à la liste `Controller`.

Une fois l'objet créé, on peut utiliser la méthode la plus importante qui permet de récupérer les informations du Leap Motion : `Frame frame(int history = 0) const`. Cette méthode renvoit par défaut la dernière trame venant du capteur. Cette méthode stocke également les 60 trames passées en modifiant l'argument de la méthode.

La classe `Listener` contient les méthodes qui sont appelées par la classe `Controller`. Ces méthodes sont appelées notamment lors de l'initialisation, la connexion et la déconnexion du Leap Motion. Une autre méthode particulièrement utile est la méthode `onFrame(const Controller&)`. C'est dans cette méthode que s'effectuera le rafraîchissement des données (trames). Elle présente une boucle intrinsèque, que l'on peut considérer comme un thread.

L'implémentation du code se fera donc dans une nouvelle classe nommée `SampleListener` qui héritera de la classe `Listener`. On pourra ainsi surcharger les méthodes existantes et en créer de nouvelles.

III. Réalisation

1. Bras robot

a. Contrôle via la console

Nous avons commencé par nous intéresser à la carte et au bras, afin de nous les approprier et d'apprendre à les contrôler.

Après s'être familiarisés avec le robot, nous avons étudié les solutions permettant la communication entre l'ordinateur et la carte. Nous avons alors élaboré un programme Arduino que nous avons téléversé sur la carte qui interprète les instructions que nous lui envoyons par la liaison série. Ces instructions prennent la forme d'une trame dont nous spécifions ci-dessous la composition.

&	1	1	0	0
début de trame	Numéro de Servo	Valeur d'angle à donner au servomoteur		

Ainsi la trame ci-dessus positionne le servo 1 à 100°. A noter qu'une valeur non comprise entre 0 et 180° induit une erreur qui repositionne le robot à une position définie que nous qualifierons de « position de repos » car cette dernière ne nécessite pas d'effort pour être maintenue. Il en est de même pour un numéro de servo non compris entre 1 et 6.

Notre programme Arduino a donc pour rôle de recevoir cette trame et de l'interpréter. Son fonctionnement est donné par la machine à état donnée en annexe 1.

Pour tester le fonctionnement du programme et les réactions induites, ainsi que le calibrer, il a été nécessaire de conduire une série de tests. Ces tests seront détaillés dans la partie adéquate. Pour les réaliser nous avons utilisé un programme sous forme de terminal permettant d'écouter et d'envoyer des données sur la liaison série. Ce programme sur Linux s'appelle *Minicom*. Son installation sur une distribution comme Ubuntu est simple, il suffit d'ouvrir un terminal et d'écrire :

```
$ sudo apt-get install minicom
```

Pour Windows, le programme **HyperTerminal** fonctionne de manière analogue.

Comme on peut le voir sur les figures suivantes, la carte répond en renvoyant les caractères envoyées (en excluant le caractère de début de trame '&') puis complète par « Fin de Trame » une fois qu'une trame complète a été transmise.

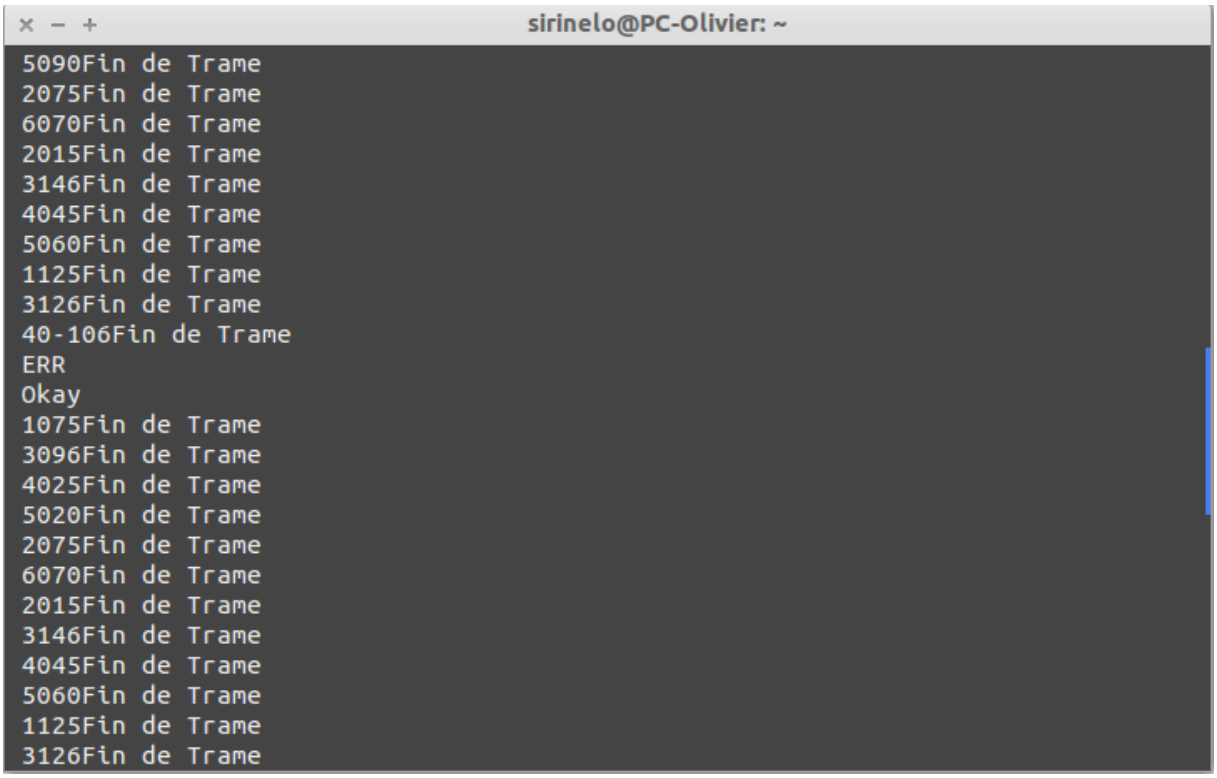


Figure 5 : Trames échangées avec la carte Arduino

Une fois le programme Arduino testé, on pouvait passer à l'implémentation de la liaison série via Qt et à l'interface graphique.

b. Contrôle par la fenêtre

Etablissement de la connexion avec le robot

Cette partie a été effectuée en parallèle avec l'élaboration de la fenêtre graphique. La liaison série est initialisée dans le constructeur de la fenêtre utilisant QextSerialPort et le code minimal fourni dans la partie conception.

Mise en place de la fenêtre :

Concernant la partie interface graphique, nous avons donc réalisé le projet en codant en C++ sous l'IDE Qt nous permettant ainsi d'utiliser de nombreuses librairies déjà toutes prêtes et notamment des librairies de Widgets pour créer des interfaces graphiques.

Le point de départ a été de créer une fenêtre en tant que telle, comme expliqué dans la partie conception. Nous avons alors obtenu une fenêtre de ce type :

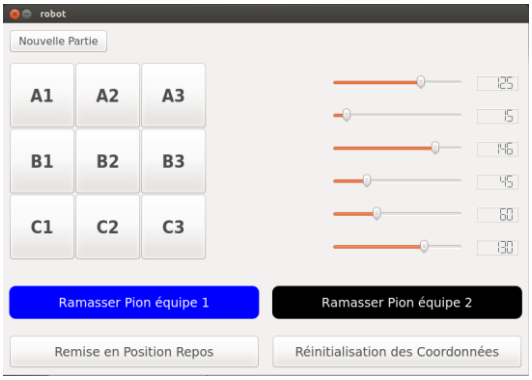


Figure 6 : interface graphique du jeu de morpion

Rendre la fenêtre fonctionnelle :

A ce point-là de la conception, la fenêtre n'est pas du tout fonctionnelle car les boutons et autres widgets n'ont aucune fonctionnalité et ne sont reliés à aucune tâche. Il a donc fallu créer des *slots*, c'est-à-dire des méthodes (ou plus simplement des fonctions) permettant d'effectuer les actions souhaitées. Ensuite ces *slots* sont connectés à des signaux émis lors d'actions sur les widgets. Par exemple, lorsque l'on déplace le curseur du *slider*, un signal est envoyé et va donc appeler le *slots* qui va modifier la valeur affichée sur l'afficheur LCD (à droite du *slider*) pour que celle-ci corresponde bien. Ou encore, lorsque l'on clique sur le bouton « A1 », un signal est émis et va appeler le *slot* donnant l'ordre au robot de déposer le pion sur la case A1. Mais à ce stade-là, le robot n'est pas encore relié à l'interface graphique. Ainsi, tout ordre donné au robot correspond juste à un changement de coordonnées envoyées sur les *sliders* de l'interface graphique.

Mise en place des règles du jeu :

Le jeu va être régulé par l'interface graphique, ainsi, bien que chaque bouton ait une fonctionnalité, il faut appliquer des règles derrière afin qu'une partie puisse bien se dérouler et qu'il ne puisse pas y avoir tricherie (ou du moins réduire ce risque au maximum) et que le joueur puisse suivre le déroulement de la partie correctement. Il a donc été nécessaire de modifier les *slots* que nous avons créés pour imposer les règles du jeu aux utilisateurs.

Ainsi, lorsqu'un pion aura été récupéré, le joueur devra impérativement déposer son pion sur une case, et c'est un joueur de l'autre équipe qui devra ramasser un pion, sinon une fenêtre contenant un message d'erreur s'affichera à l'écran. De plus, lorsqu'un pion sera déposé sur une case du plateau de jeu, celle-ci s'affichera de la couleur du pion déposé et sera désactivée, c'est-à-dire qu'il sera impossible de déposer un autre pion à cet endroit. Ainsi, pour une partie entamée, la fenêtre de jeu ressemblera à ceci (cf figure suivante) :

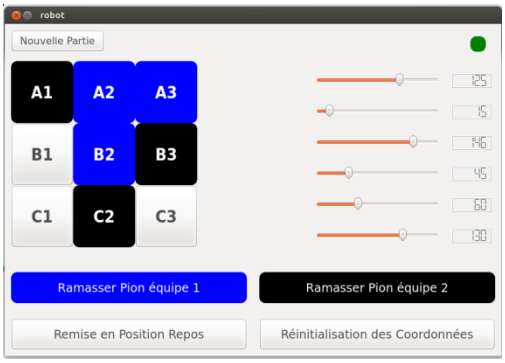


Figure 7 : interface graphique du jeu de morpion en pleine partie

Liaison de l'interface graphique avec le bras de Robot :

La création du lien entre le robot et l'interface graphique a également été développée sous Qt. Elle utilise ici encore des librairies spéciales de Qt permettant de faciliter son utilisation.

La liaison se fait ici encore à l'aide de *slots* qui ont été créés pour faire en sorte que lorsque l'on bouge un *slider*, une donnée de type adaptée soit envoyée au robot via liaison série. Par exemple, lorsqu'on modifie la valeur du *slider 1* pour lui donner un angle de 45°, l'information envoyée au robot est de la forme « &1045 » selon la syntaxe que nous avons défini précédemment. Cet envoi est réalisé par une méthode présente dans la classe *MaFenetre* qui gère l'interface graphique.

Ainsi, le contrôle du robot se fait par six fonctions différentes appelées dès qu'un *slider* a été modifié et envoyant l'ordre directement au robot via port série.

De plus, afin de vérifier que le robot est bien connecté à l'interface graphique via port-série, nous avons mis en place un voyant en haut de la fenêtre graphique. Celui-ci sera vert si le robot est bien connecté et rouge si la connexion n'est pas établie (un message d'erreur sera également affiché afin d'expliquer la procédure à suivre et pour les éventuelles personnes souffrant de daltonisme qui voudraient jouer au morpion avec notre interface graphique).

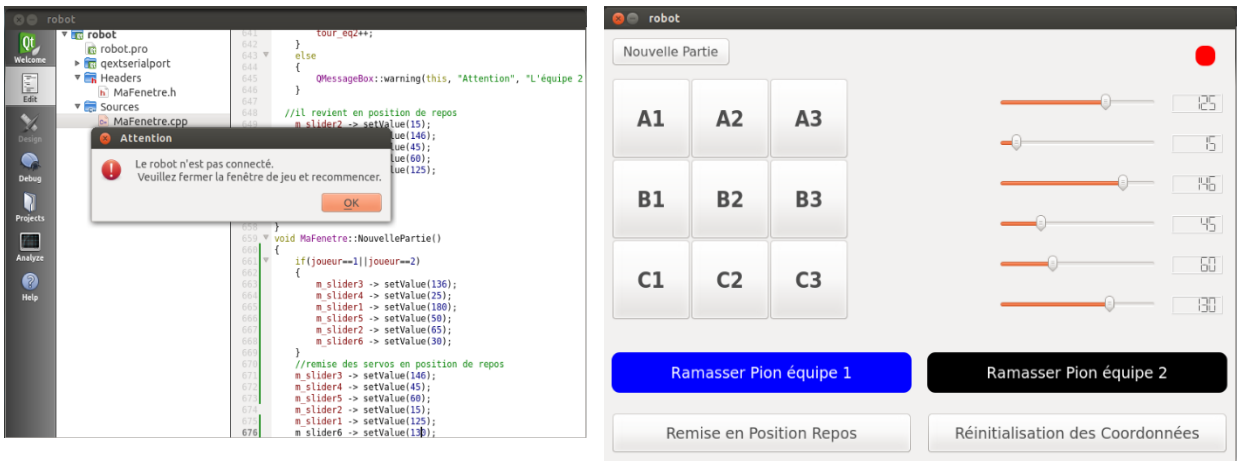


Figure 8 : Message d'erreur et fenêtre montrant que le robot n'est pas connecté

Principales difficultés rencontrées :

La mise en place des Widgets ainsi que leur disposition sur la fenêtre et la création des slots en tant que tels n'a pas été très difficile. La principale difficulté a été de respecter l'enchaînement des joueurs et de savoir à quel tour de la partie nous nous trouvions. En effet, pour éviter la tricherie, il a fallu créer une variable *joueur* permettant de déterminer quel joueur doit jouer et si oui ou non il a déjà tiré un pion. En effet, quand arrive le moment de ramasser un pion, il faut que le robot ait déposé le pion qu'il avait ramassé au tour d'avant et que ce ne soit pas la même équipe que celle qui vient juste de jouer. Pour cela, nous avons choisi d'attribuer 5 valeurs possibles à notre variable *joueur* : une disant que l'équipe 1 vient de ramasser un pion, une autre explicitant que l'équipe 1 vient de poser un pion, et par symétrie, deux autres valeurs correspondant à l'équipe 2. La cinquième valeur possible est donnée lorsque l'on réinitialise la partie et qu'aucun pion n'a encore été pris ou déposé sur le plateau de jeu.

Une autre difficulté a été de gérer le fait qu'à chaque tour, le robot ne doit pas aller chercher le pion au même endroit. En effet, le plateau de jeu est comme sur la figure ci-dessous, on peut voir que les pions sont posés les uns à la suite des autres en attendant d'être ramassés. Il faut donc créer une variable pour chaque équipe permettant de déterminer combien de pions ont été posés par l'équipe pour savoir où aller chercher le suivant.

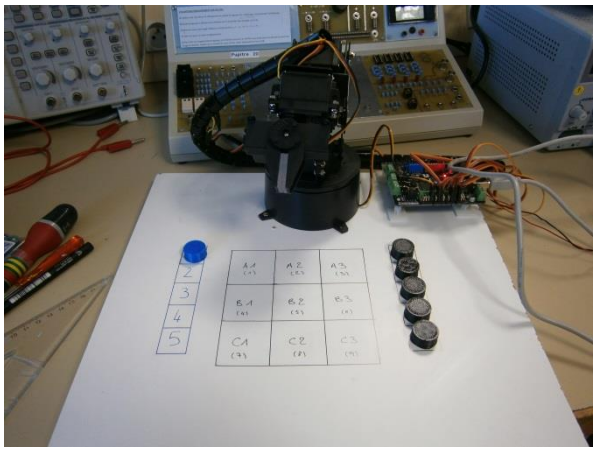


Figure 9 : photo du plateau de jeu réel (avec le robot)

Enfin, le travail le plus long lié à cette partie du projet, a été de déterminer les angles que l'on devait donner à chaque servomoteur du bras pour se déplacer sur le plateau de jeu. En effet, n'ayant pas eu le temps de trouver les équations de cinématique inverse, il nous a fallu déterminer de manière empirique la position de chaque servomoteur pour chaque position possible du robot, soit pour les 9 cases du jeu plus les 10 positions correspondant au ramassage des pions sur le côté. De plus, un des principaux problèmes rencontrés face à cela est le manque de répétabilité des mouvements par le bras de robot. En effet, les servomoteurs sont plutôt imprécis et il est difficile de réussir à obtenir du bras de robot de se placer plusieurs fois exactement au même endroit sur le plateau.

Enfin, une précaution fondamentale qu'il a fallu prendre, est de faire bien attention à ce que les ordres envoyés au robot se fassent dans un ordre précis afin que le bras ne balaye pas tous les pions déjà déposés lors de son passage. Par exemple, si une fois que le pion a été ramassé, le robot pivote d'abord son premier servomoteur relié à la base pour tourner alors que la pince est contre le plateau, tous les pions qui étaient au départ sur le plateau de jeu sont déplacés par son mouvement. Il est donc très important de respecter un ordre précis comme par exemple de commencer par se redresser puis ensuite tourner pour se placer dans la bonne direction.

De plus, par soucis d'esthétique et de limitation de « gestes » trop brusques du robot, après chaque mouvement (ex : ramasser pion, ou déposer pion en A2), le robot se remettra en position de repos en attendant d'avoir connaissance de l'ordre suivant. Cela ajoute un délai supplémentaire pour le jeu (car impose des mouvements en plus) mais protège le robot de certains pics de consommation de courants supplémentaires qui pourraient l'endommager.

2. Leap Motion

a. Contrôle du robot depuis le Leap Motion pas à pas

i. Description de la réalisation

L'objectif de cette partie, est de transmettre au robot le numéro du servomoteur à bouger ainsi que son angle de rotation.

Pour cela, nous choisissons la convention suivante pour l'utilisateur :

- Le nombre de doigts de la main gauche visibles pour le Leap Motion code le numéro du servomoteur.
- La main droite quant à elle ne doit avoir qu'un doigt visible (par conséquent l'index pour une question ergonomique). La hauteur de l'index par rapport au Leap Motion est alors proportionnelle à l'amplitude de l'angle de rotation du servomoteur en question.

ii. Description de l'architecture de notre code

Attributs ajoutés à la classe `SampleListener`

Nous avons créé trois structures contenues dans le fichier Structures.h.

- Une structure INFOS_LEAP :

```
typedef struct {
    int nbr_doigts ;
    float x ;
    float y ;
    float z ; } INFOS_LEAP;
```

Où `nbr_doigts` correspond au nombre total de doigts visibles par le Leap Motion, qui est en fait le numéro du servomoteur à contrôler. En effet, quand la main gauche ne montre aucun doigt, l'index de la main droite, lui, est toujours visible. Dans ce cas, le nombre de doigts total visibles est de 1, c'est donc le servomoteur numéro 1 qu'il faut contrôler. En résumé, 0 doigt visible pour la main gauche correspond au servo numéro 1, 5 doigts visibles sur la main gauche indiquent le contrôle du servo moteur numéro 6.
Les flottants x, y et z correspondent aux coordonnées de l'index de la main droite.

- Une structure INFOS_ROBOT :

```
typedef struct {
    int servo ;
    int angle ; } INFOS_ROBOT;
```

Cette structure contient les informations à transmettre au robot. C'est-à-dire le numéro du servomoteur à bouger, soit servo, et l'amplitude de son angle, soit angle.

- Une structure SERVO :

```
typedef struct {
    int numero;
    int a_min;
    int a_max ; }   SERVO;
```

La structure SERVO contient toutes les informations utiles à la description d'un servomoteur. C'est-à-dire son numéro, soit le champ `numero`, et les angles maximum qu'il peut atteindre, soit `a_min` et `a_max`. En effet tous les servomoteurs n'ont pas une amplitude de 0 à 180 degrés.

Nous avons alors complété la classe `SampleListener` de 3 attributs supplémentaires qui sont :

- Les informations sur la commande passées par les doigts de l'utilisateur : `INFOS_LEAP infos_leap ;`
- Les informations à transmettre au robot : `INFOS_ROBOT infos_robot;`
- Un tableau contenant les 6 servomoteurs : `SERVO tab_servo[6];`

Méthodes ajoutées à la classe SampleListener

De manière à organiser notre code au maximum, nous avons créé 5 méthodes dans la classe SampleListener :

```
virtual void init_tab_servo();
```

Cette méthode permet d'initialiser le contenu du tableau de servo.

```
virtual int mains(const Controller&);
```

La méthode `mains` permet de différencier la main droite de la main gauche. En effet, le Leap Motion attribue un numéro aux mains dans leur ordre d'apparition. Pour ne pas rajouter de convention au joueur (par exemple : avancer la main droite en premier), il est nécessaire de connaitre le numéro attribué à la main droite pour ensuite récupérer les coordonnées de l'index contrôlant l'amplitude de l'angle du servomoteur.
Si deux mains sont effectivement visibles par le Leap Motion, alors la méthode distingue la main droite de la main gauche par le calcul de la position moyenne en x des doigts.
Si une ou plus de deux mains sont détectées par le capteur, un message avertit l'utilisateur.

```
virtual void infos_doigt_commande(const Controller&,int
main_d);
```

La méthode *infos_doigt_commande* complète la structure attribut infos_leap de la classe *SampleListener*. Elle remplit également le champ servo de la structure attribut *infos_robot*.

```
virtual void calcul_angle(const Controller&);
```

Comme son nom l'indique, la méthode calcul_angle calcule l'angle à renvoyer au servomoteur à partir de la coordonnée en z de l'index de la main droite.
Le calcul se fait de la manière suivante :
Nous avons déterminé des amplitudes extrêmes qui sont d'une part détectables par le Leap Motion et d'autre part ergonomique pour l'utilisateur. Ces amplitudes sont stockées dans les variables globales Z_MAX et Z_MIN (ici, respectivement 400 et 100 millimètres). L'angle maximal atteignable par le servo désigné correspond à Z_MAX et l'angle minimal à Z_MIN. A partir de ces liens et l'amplitude de l'index droit, nous pouvons facilement déterminer l'angle à transmettre.

```
virtual INFOS_ROBOT get_infos_robot();
```

La méthode get_infos_robot est un accesseur permettant de retourner l'attribut infos_robot contenant les informations utiles au contrôle du bras robot.

Modification de la méthode onFrame de la classe SampleListener

```
virtual void onFrame(const Controller&);
```

Cette méthode est appelée dans le *main* et tourne en boucle, c'est donc elle qui traite directement les données envoyées par le Leap Motion. Ainsi, nous l'avons modifiée pour qu'elle exécute notre algorithme.
onFrame commence par appeler la méthode mains pour distinguer la main droite de la main gauche.
Si 2 mains ont bien été détectées par la méthode *mains* alors *onFrame* appelle d'une part la méthode *infos_doigt_commande* et d'autre part la méthode *calcul_angle*.

Contenu du main

Lors de l'exécution du main, celui-ci crée des objets controller et listener puis initialise le tableau de servo par l'appel de la fonction *listener.init_tab_servo()*. Ensuite la méthode onFrame est appelée par l'objet controller et tourne en boucle tant que l'utilisateur n'appuie pas sur la touche "entrée".

iii. Test de rafraîchissement des données du Leap Motion

Le site du Leap Motion annonce une vitesse de rafraichissement d'environ 200 Hz. Or les mesures qui ont été réalisés dans la méthode onFrame ne sont pas en accord avec ces chiffres. Les mesures ont été réalisées en écrivant un fichier sans affichage à l'écran. Voici le résumé des mesures en Hz :

Valeur minimale : 23.8095 Hz
Valeur maximale : 32.2581 Hz
Valeur moyenne : 27.8167 Hz
Ecart type : 1.1537 Hz

iv. Difficultés de réalisation rencontrées.

Nous avons passé un temps certain à comprendre le fonctionnement des méthodes déjà codées dans le SDK. En effet, seul le fichier sample.cpp et les fichiers .h sont visibles. Non n'avons donc pas eu accès directement au code des méthodes, les autres fichiers .cpp étant protégés. Ceci ne nous a pas permis de comprendre précisément le rôle de chaque classe.

b. Contrôle du bras-robot par une grille de morpion virtuelle

Ayant créé une interface graphique pour contrôler le bras-robot. Nous avons trouvé intéressant de développer un autre programme qui permettrait à l'utilisateur de jouer au morpion depuis une grille virtuelle.

i. Description de la réalisation

Nous avons constaté que lorsque nous collions deux de nos doigts, le capteur ne détecte alors aucun doigt. Nous avons donc basé notre algorithme sur cette constatation. En effet, le joueur sera capable de sélectionner une case en passant de deux doigts écarté à deux doigts collés. Le Leap Motion traduit cela comme un passage de deux à zéro doigt.

Pour commencer une partie, le joueur devra d'abord définir la localisation de sa grille virtuelle dans l'espace en sélectionnant en premier l'angle en haut à gauche puis l'angle en bas à droite. Par convention la grille de morpion est définie verticalement et face au joueur, c'est-à-dire selon les coordonnées x et z.
Ensuite la partie peut commencer. La case survolée par le joueur est affichée à l'écran et le joueur referme les doigts lorsqu'il veut sélectionner la case. Cette dernière est alors stockée dans les attributs de la méthode *SampleListener* et est transmise au robot.
A tout moment, le joueur peut réinitialiser la partie en avançant ses deux mains devant le capteur.

ii. Description de l'architecture du code

Attributs ajoutés à la classe SampleListener :

Cette fois, nous avons ajouté une structure au fichier Structures.h :

- La structure CASE :

```
typedef struct {
   float x_max;
   float x_min;
   float z_max;
   float z_min;
   int numero;} CASE;
```

Cette structure regroupe tous les paramètres permettant de décrire une case. C'est-à-dire, son numéro par le champ `numero` et l'espace qu'elle occupe par les champs `x_max`, `x_min`, `z_max`, `z_min`.

Nous avons donc complété la classe `SampleListener` avec les attributs suivants :

- Un tableau contenant les 9 cases de la grille de morpion : `CASE tab_cases[9];`
- Le numéro de l'étape en cours suivant la représentation schématique de l'algorithme (voir plus loin) : `int etape;`
- Un entier représentant le numéro de la case survolée pendant le jeu : `int num_case_current;`
- Un entier représentant le numéro de la case sélectionnée (C'est cet attribut qui est envoyé au robot) : `int num_case_select;`
- Des flottants permettant de stocker les coordonnées de l'angle en haut à gauche (x1, z2), les coordonnées de l'angle en bas à droite (x2,z1) et les coordonnées des points sélectionnés pendant le jeu (x,z) : `float x1, x2, z1, z2, x, z;`
- Un entier représentant le nombre de doigts vu à la boucle précédente : `int p_nbr_d;`

Méthodes ajoutées à la classe SampleListener :

```
virtual void init_etape(int a);
```

La méthode `init_etape` permet de mettre à jour l'attribut etape.

```
virtual void init_tab_cases(float x1, float x2, float y1,
   float y2);
```

La méthode `init_tab_cases` initialise le tableau des cases à partir de coordonnées des angles de la grille.

```
virtual void init_coor();
```

La méthode `init_coor` met des valeurs improbables aux attributs coordonnées de manière à s'assurer que ces dernières ont été modifiées au moins une fois avant de les utiliser pour les calculs.

```
virtual int get_case_current(float x, float y);
```

La méthode `get_case_current` affiche à l'écran le numéro de la case survolée par l'utilisateur à partir des moyennes de la position de ses deux doigts en x et z. Il stocke également ce numéro dans l'attribut `num_case_current`.

```
virtual int get_case_select(float x, float y);
```

La méthode `get_case_select` fonctionne sur le même principe que la méthode `get_case_current` mais stocke le numéro dans l'attribut `num_case_select`.

```
virtual float average_x(FingerList fingers);
```

La méthode `average_x` calcule la position moyenne en x des doigts fournis en argument.

```
virtual float average_z(FingerList fingers);
```

La méthode `average_z` calcule la position moyenne en z des doigts fournis en argument.

```
virtual void init_p_nbr_d();
```

La méthode `init_p_nbr_d` permet d'initialiser l'attribut p_nbr_d à zéro.

```
virtual int getCaseSelect();
```

La méthode `getCaseSectect` est un accesseur qui permet de récupérer la valeur de l'attribut `num_case_select`.

Modification de la méthode onFrame de la classe SampleListener :

La méthode `onFrame` a été modifiée de manière à implémenter l'algorithme représenté par le schéma ci-dessous :

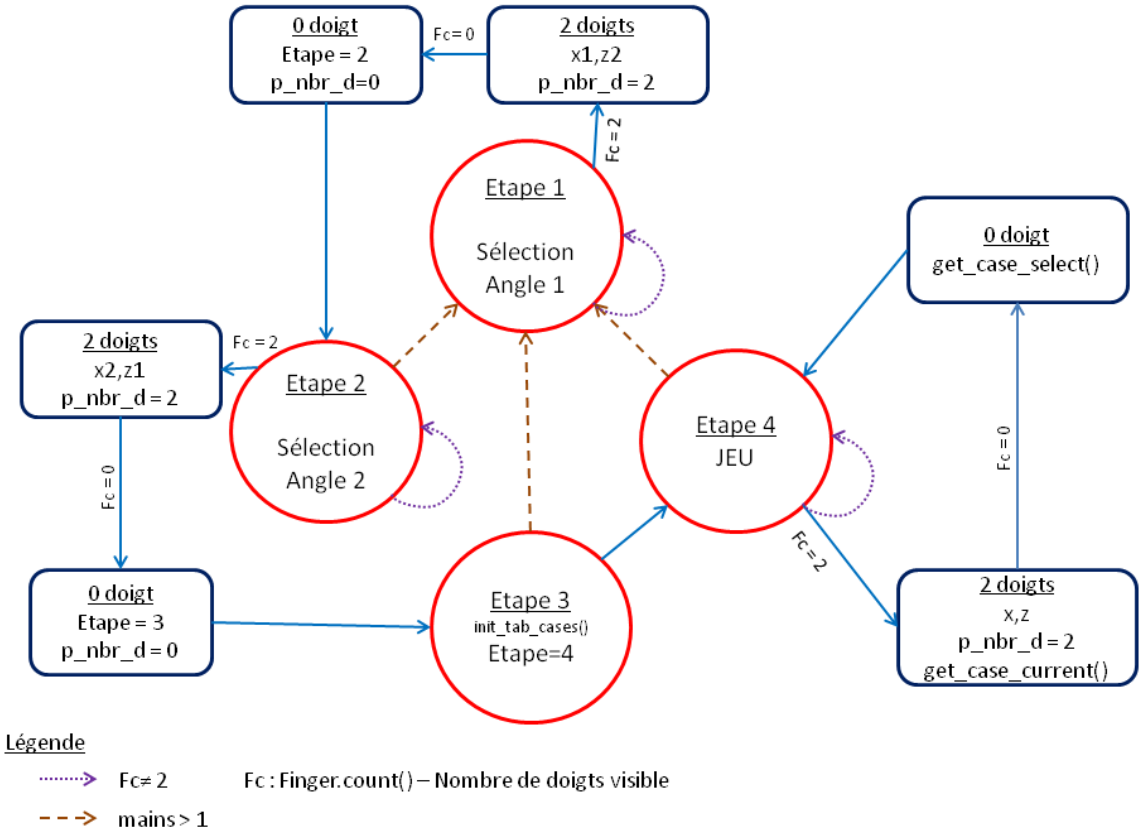


Figure 10 : Schéma de principe

Contenu du main :

Lors de l'exécution du `main`, celui-ci crée des objets `controller` et `listener` puis appelle les méthodes d'initialisation : `init_p_nbr_d`, `init_etape`, `init_coor`. Ensuite le main appelle la méthode `onFrame` qui tourne en boucle tant que l'utilisateur n'appuie pas sur la touche "entrée".

iii. Difficultés de réalisation rencontrées.

Suite à l'implémentation du premier programme, nous n'avons pas eu de mal à reprendre en main les méthodes déjà codées dans le SDK. La majeure difficulté de ce programme a été l'élaboration de l'algorithme. En effet, chaque étape nécessite beaucoup de conditions qui ont entrainé de nombreuses reprises du code pour être sûr qu'aucune situation n'ait été oubliée.

Ce système a finalement bien été implémenté, mais dans un second programme où l'interface graphique a été supprimée. En effet, il existe un problème de communication entre les deux éléments, problème que nous n'avons pu résoudre. Néanmoins il est malgré tout parfaitement possible de jouer au morpion avec ce programme.

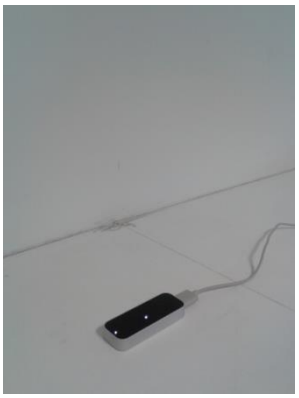
IV. Modalités de validation

1. Validation du programme Leap permettant de contrôler le bras robot pas à pas

Pour valider le fonctionnement de l'algorithme permettant de jouer au morpion avec une grille virtuelle, nous avons choisi de présenter l'affichage sur la console en parallèle de la position des mains associées en photo.

- Aucun contrôle si aucune main.

```
Valeur de la variable mains : 0
Aucun servo moteur n'est designe
```

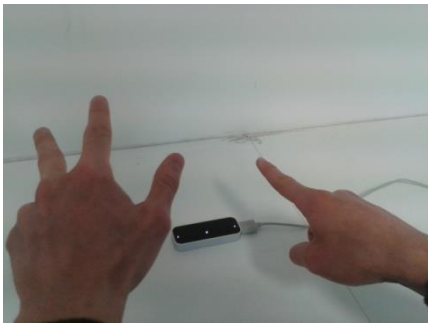


- Trois doigts à gauche signifient contrôle du servomoteur 4.

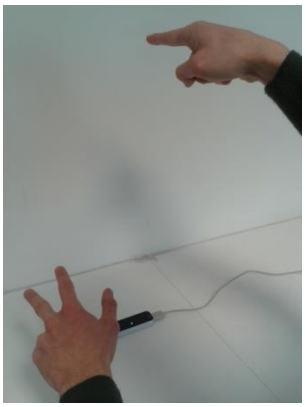
```
Valeur de la variable mains : 2
Position moyenne main gauche : -100.264 Position moyenne main droite : 55.1339

Servo a controller : 4 Position du doigt de commande : 155.01

Numero du servo : 4 Angle  : 20
```

- Doigt à droite plus bas que précédemment, l'angle a diminué de moitié (passant de 20 à 10).

```
Valeur de la variable mains : 2
Position moyenne main gauche : -98.8405 Position moyenne main droite : 52.2075

Servo a controller : 4 Position du doigt de commande : 96.353

Numero du servo : 4 Angle  : 10
```

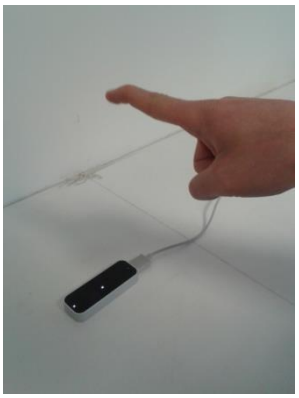


- Doigt à droite plus haut que précédemment mais au-dessus de Z_Max (400), ici 444. L'angle calculé reste égal à la valeur maximale de l'angle atteignable par le servomoteur 4.



```
Valeur de la variable mains : 2
Position moyenne main gauche : -121.571 Position moyenne main droite : 30.5301

Servo a controller : 4 Position du doigt de commande : 444.304

Numero du servo : 4 Angle  : 65
```

- Si une seule main, aucun ordre passé au robot.



```
Valeur de la variable mains : 1
Aucun servo moteur n'est designe
```

- Message d'erreur si plus de deux mains.

```
Valeur de la variable mains : 4
Un humain normalement constitue ne possede pas plus de deux mains, Tricheur !
```

- Message d'erreur si plus d'un doigt à droite.

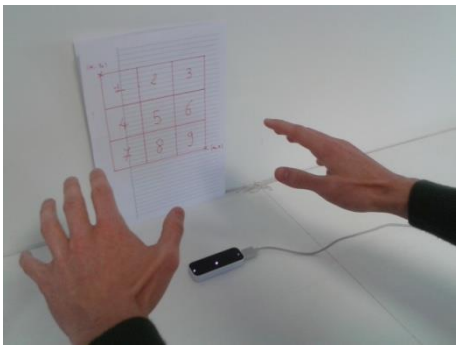
```
Valeur de la variable mains : 2
Position moyenne main gauche : -146.008 Position moyenne main droite : 40.5689

Servo a controller : 10 Position du doigt de commande : 154.603
Position de la main non standardise (Un doigt a droite)
```

2. Validation de l'algorithme Leap permettant de jouer avec une grille virtuelle

Pour valider le fonctionnement de l'algorithme permettant de jouer au morpion avec une grille virtuelle, le principe est le même.

- Deux mains détectées.

```
Nombre de main : 2
Reinitialisation du jeu
```

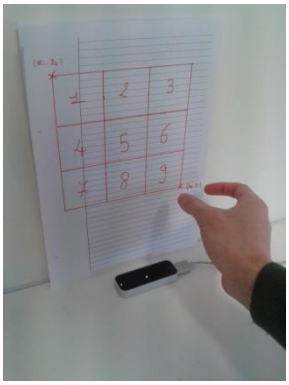


- Sélection du premier angle.

```
Nombre de main : 1
Angle en haut a gauche ! x1 : -67.548 z2 : 189.461
```

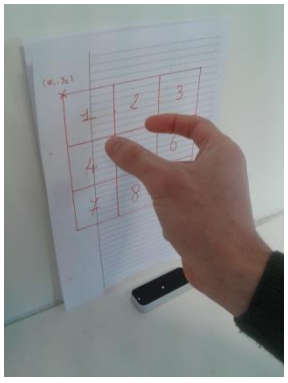
- Sélection du deuxième angle.

```
Nombre de main : 1
Angle en bas a droite ! x2 : 41.7523 z1 : 75.8348
```



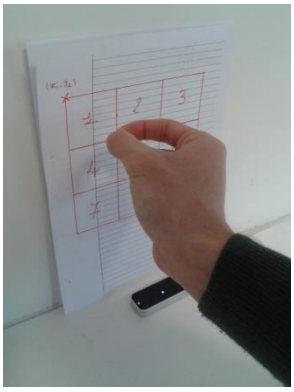
- Attente du choix de la case.

```
Nombre de main : 1
Non fixes - x : -46.7755 z : 168.698
Case survolee : 1
```



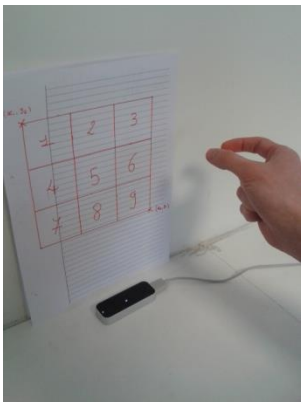
- Sélection de la case.

```
Nombre de main : 1
Case selectionnee : 1
```



- Cas où la main sort de la grille.

```
Nombre de main : 1
Non fixes - x : 48.2635 z : 143.559
Ce point n'appartient pas au damier
```



3. Validation bras-robot

Au terme de ce projet, l'interface graphique reliée au robot est totalement fonctionnelle. Une partie de morpion peut être jouée et se dérouler de manière très satisfaisante (bien que prenant plus de temps que si les joueurs jouaient à la main mais permettant de ne pas avoir à déplacer les pièces soi-même). Le déroulement du jeu se déroule notamment sans tricherie possible de la part des joueurs. Il faut cependant à la fin de chaque partie que les joueurs remettent eux-mêmes les pions dans l'espace prévu à cet effet afin de pouvoir recommencer une nouvelle partie.

En revanche, la fonctionnalité permettant de réinitialiser les coordonnées des cases n'a pas été implémentée. Le code est alors figé et ne permet de jouer que sur le plateau qui a été conçu à cet effet et ne permet pas la mobilité de jeu. Cela est dû au fait que, par manque de temps, nous n'ayons pas pu mettre en place les équations de cinématique inverse.

Sur la vidéo de présentation du projet, vous trouverez une démonstration du fonctionnement complet du bras robot, piloté par l'interface graphique et par le Leap Motion.

V. Apports personnels

Fanny :

Pour ma part, je me suis principalement intéressée au développement du code pour implémenter l'interface graphique. Ayant choisi de développer cette fenêtre en C++ sous l'IDE Qt, il a donc été nécessaire de comprendre comment fonctionne la programmation orientée objet (car quand nous avons commencé le projet nous n'avions pas encore commencé les cours de java) et apprendre la syntaxe du C++. Ce projet m'a donc permis d'apprendre beaucoup en termes de compétences en programmation.

Par ailleurs, coder une interface graphique est très agréable car il est possible à tout moment d'avoir un aperçu concret de ce que nous sommes en train de mettre en place, ce qui est rarement le cas (du moins au cours de notre formation en école). De plus, être dans le cadre de la mise en place d'une interface graphique pour un jeu de morpion avait vraiment un aspect ludique qui rendait le projet bien plus motivant.

Au cours de ce projet, nous sommes vraiment partis de quasiment rien pour arriver à un système fonctionnel permettant de jouer au morpion avec un bras de robot. Nous avons ainsi pu mener une démarche de conception quasiment de A à Z. Nous avons dû faire la quasi-totalité des choix techniques afin de décider ce qui était le mieux pour nous et pour le bon déroulement du projet et rédiger un véritable cahier des charges pour notre système. Cela a alors été très instructif et reflète réellement ce qu'est le métier d'un ingénieur.

Pour finir, nous travaillions en équipe de 4, et nous avons ainsi pu voir l'importance de bien séparer les tâches tout en se tenant au courant de l'avancée de chacun dans le projet. En effet, il nous aurait été impossible d'obtenir un système fonctionnel si nous ne nous étions pas concentrés chacun sur une tâche bien particulière du projet.

Pour conclure, je peux donc dire que ce projet me paraît vraiment très important dans notre cursus d'ingénieur car il permet vraiment de nous donner une idée de ce dont va être fait notre futur métier. Je pense que c'est un très bon exercice qui nous permet d'apprendre bien plus que si nous avions eu des cours à la place car on apprend, selon moi, bien plus en pratiquant et en étant vraiment confronté à un problème réel et concret.

Olivier :

De mon côté, j'ai travaillé de concert avec Fanny pour le fonctionnement du robot via l'interface graphique. De ce fait, j'ai dû apprendre un nouveau langage de programmation le C++ et à travailler avec une API. C'est d'ailleurs une expérience assez étrange, on manipule des objets dont on ne connaît pas du tout le fonctionnement.

Tandis que Fanny nous programmait une magnifique interface graphique, j'ai été en charge de l'interfaçage avec le robot, en d'autres termes je devais m'assurer que les données envoyées depuis le programme principal étaient bien envoyées sur la liaison série, reçues et correctement interprétées par le robot. Pour cela j'ai dû manipuler une librairie inconnue d'une API que je découvrais, inutile de dire qu'au premier abord c'est assez déroutant mais ô combien formateur sur le plan de la programmation. La documentation devient alors une seconde bible.

Devant un projet aussi vaste, en dehors du choix de la carte et du bras robot nous avons tout réalisé, le travail d'équipe devient indispensable. Nous étions un groupe dynamique et nous nous entendions bien, cela a simplifié les choses. La division des tâches et l'efficacité de chacun a permis d'arriver à un résultat que nous espérons satisfaisants.

Enfin, ce projet avait un grand frère, un projet réalisé avec le même matériel par un groupe de SMPB 3A. Nous avons pour commencer pris connaissance de leur travail, qui nous a aidés à nous familiariser avec Arduino. J'ai alors réalisé que si notre projet devait être continué, nous devions réaliser un rapport le plus complet et le plus exhaustif complet pour permettre à nos potentiels

successeurs de partir d'un bon pied, cela vaut aussi pour le code que nous espérons suffisamment commenté.

En conclusion, et bien que ce projet soit plus court qu'un projet ingénieur classique, il nous a permis de faire face à des problèmes concrets de conception, tout en nous donnant un retour très visuel sur notre travail.

Astrid :

Pour ce projet, je me suis plus particulièrement intéressée à la partie réalisation rattachée au Leap. Appréhender un nouveau matériel dont on ne connait rien est un peu déroutant au premier abord. Il a fallu du temps pour se familiariser avec le code fourni et ainsi comprendre de quelle manière concevoir nos propres algorithmes. Cependant cette démarche est très enrichissante. De plus le fait d'aboutir à deux types de programmes pour contrôler le bras-robot qui fonctionnent de manière indépendante est très satisfaisant.

Concernant le travail en équipe, notre groupe était divisé en deux puisque le projet présentait deux parties bien distinctes. De ce fait, nous devions nous tenir régulièrement au courant de l'avancement des sous équipes en organisant des réunions à chaque début de séances. Ce fonctionnement a renforcé mon habilité à occuper correctement ma place au sein d'une équipe de travail.

Pierre :

Tout d'abord d'un point de vue technique, ce projet était essentiellement orienté vers la programmation. Le langage auquel nous étions confrontés étant le C++, nous pouvions établir des liens conjointement avec le Java étudié en cours. Mon travail était de travailler sur la compréhension du code provenant des librairies du Leap Motion ainsi que de développer le code d'interprétation des mouvements. On pourra reconnaitre qu'il n'est pas très agréable de travailler sur un produit dont le code est protégé car cela ralentit grandement la facilité de s'approprier le code. Néanmoins, le Leap Motion était un capteur intéressant à manipuler car les tests pouvaient être réalisés très rapidement. D'autre part, la division du travail en 2 équipes s'est avérée assez efficace. À l'intérieur de chaque équipe, un membre travaillait à l'interfaçage des deux parties ce qui a rendu la fusion du projet plus efficace.

Sur le plan relationnel, j'ai trouvé le travail avec les autres membres de l'équipe très agréable. Nous étions très motivés pour travailler sur ce projet et l'ambiance s'en ressentait au sein de l'équipe.

Conclusion : Amélioration possible du projet

Partie interface graphique :

Dans l'état actuel des choses, le robot est totalement fonctionnel, cependant, au niveau du code, les widgets sont disposés de manière absolue sur la fenêtre. Cela implique donc que si un développeur veut changer la taille de la fenêtre pour l'agrandir ou autre, les widgets resteront à leurs emplacements actuels et n'évolueront pas avec la fenêtre, ce qui peut être très dommageable s'il décide de diminuer la taille de la fenêtre, faisant ainsi disparaître les widgets. Il pourrait donc être possible de mettre en place un *layout* permettant d'avoir un positionnement relatif des widgets et de ne pas avoir les problèmes explicités.

Par ailleurs, comme nous en avons déjà parlé précédemment, il pourrait être intéressant de déterminer les équations de cinématique inverse permettant de mettre en place une plus grande portabilité du jeu (en pouvant notamment changer le plateau de jeu, autant en positions qu'en dimensions) et également de mettre en place une fonction permettant éventuellement au robot de récupérer les pions à la fin de la partie pour les « ranger » à leur place en attendant le début d'une nouvelle partie.

Enfin, une dernière amélioration possible serait de mettre des servomoteurs plus puissants et plus précis permettant une meilleure répétabilité que celle que nous avons actuellement.

Partie damier virtuel :
Permettre l'interfaçage avec la partie graphique.

Niveau compatibilité, nos deux programmes principaux sont fonctionnels sur Linux (testé sur Ubuntu 32 et 64 bits) moyennant l'installation du Leap SDK.

Pour Windows, la compatibilité avec le programme interface graphique est assurée sur des processeurs 32 et 64 Bits. Pour le programme Leap Motion en revanche, la portabilité n'est pas réalisée, c'est un point à retravailler.

Avec interface graphique

Fichier/Dossier	Rôle
Main.cpp	Lancement du programme
MaFenetre.cpp / .h	Contient l'objet MaFenetre, notre interface graphique qui gère la partie et la liaison série
Lib/	Contient la librairie QExtSerialPort

Avec le Leap Motion

Fichier/Dossier	Rôle
Main.cpp	Lancement du programme
Serie.cpp / .h	Contient l'initialisation de la liaison série et la gestion de la partie
Sample.cpp / .h	Routine du Leap Motion. Appelle la liaison série.
Include/ & LIBS	Contient les librairies du Leap Motion.

Démonstration du projet : Les liens des vidéos ainsi que d'autres ressources liées à notre projet peuvent se trouver à l'adresse suivante :

> http://sirinelli.fr/granola

Bibliographie

Manuel de la Carte :
http://www.zartronic.fr/doc/DFR/DFR0004/Zartronic_Guide_Utilisateur_RomeoV110_2012.pdf

Instruction de montage du bras ROB0036
http://www.dfrobot.com/newsletter/assembly%20guide%20for%20rob0036/Instruction_de_ROB0036.pdf

https://learn.sparkfun.com/tutorials/leap-motion-teardown/all

https://forums.leapmotion.com/forum/general-discussion/general-discussion-forum/434-the-unofficial-leap-faq?420-The-unofficial-Leap-FAQ

http://fr.lookcloseseefar.com/leap-motion-larticle-pour-tout-savoir/

https://forums.leapmotion.com/forum/general-discussion/general-discussion-forum/1058-technical-specifications

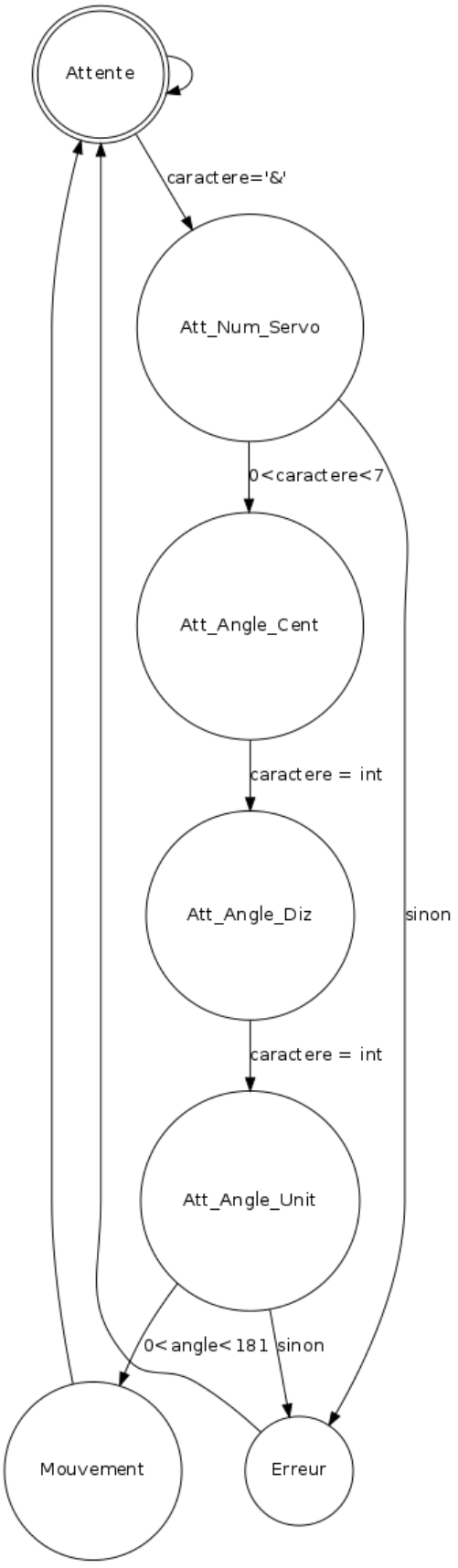
www.leapmotion.com

www.tutoprocessing.com/avance/leap-motion-processing /

www.developer.leapmotion.com/documentation/cpp/devguide/Project_Setup.html

www.fr.openclassrooms.com/informatique/cours/programmez-avec-le-langage-c

Annexe 1 : Machine à état pour le programme Arduino



Annexe 1: Machine à état représentant le programme téléversé sur la carte

Annexe 2 : Manuels d'utilisation

- **Pilotage du Robot en console**

Prérequis :

- Logiciel minicom sur Unix (sudo apt-get install minicom), Hyperterminal sur Windows
- le bras robot dûment connecté avec le programme téléversé

Régler minicom (ou HyperTerminal) de manière à avoir le port correspondant au robot sur écoute (sur Ubuntu par exemple : **/dev/ttyACM0**) et à bien envoyer des caractères ASCII (format décimal...)

On communique avec le robot avec la syntaxe suivante :

- On commence par initier une nouvelle trame avec le caractère **&**
- On entre ensuite le numéro du servo (voir schéma) : ex : 1 pour le servo de base
- On entre ensuite la valeur de l'angle que l'on veut lui donner. Ex : 120

On a donc finalement une instruction de la forme *&1120*.
Une fois le caractère **&** détecté, le robot écrit sur la liaison série les caractères reçus :
>> $ &1120 (données expédiées, non visible sur la fenêtre)
>> 120 Fin de Trame (Ce que renvoie le robot)
Il se déplace ensuite jusqu'à la position donnée par pas de 2°. En effet il est illusoire d'avoir une précision au degré avec le robot.

Ceci permet donc de contrôler le robot à la main.

- **Pilotage du Robot par interface graphique**

Prérequis :

- Avoir Qt installé (et un environnement UNIX)
- Le bras robot dûment connecté avec le programme téléversé
- **s'assurer que le robot est bien connecté sur le port /dev/ttyACM0 (seul supporté pour l'instant)**

Ci-dessous on trouvera une capture d'écran de l'interface.

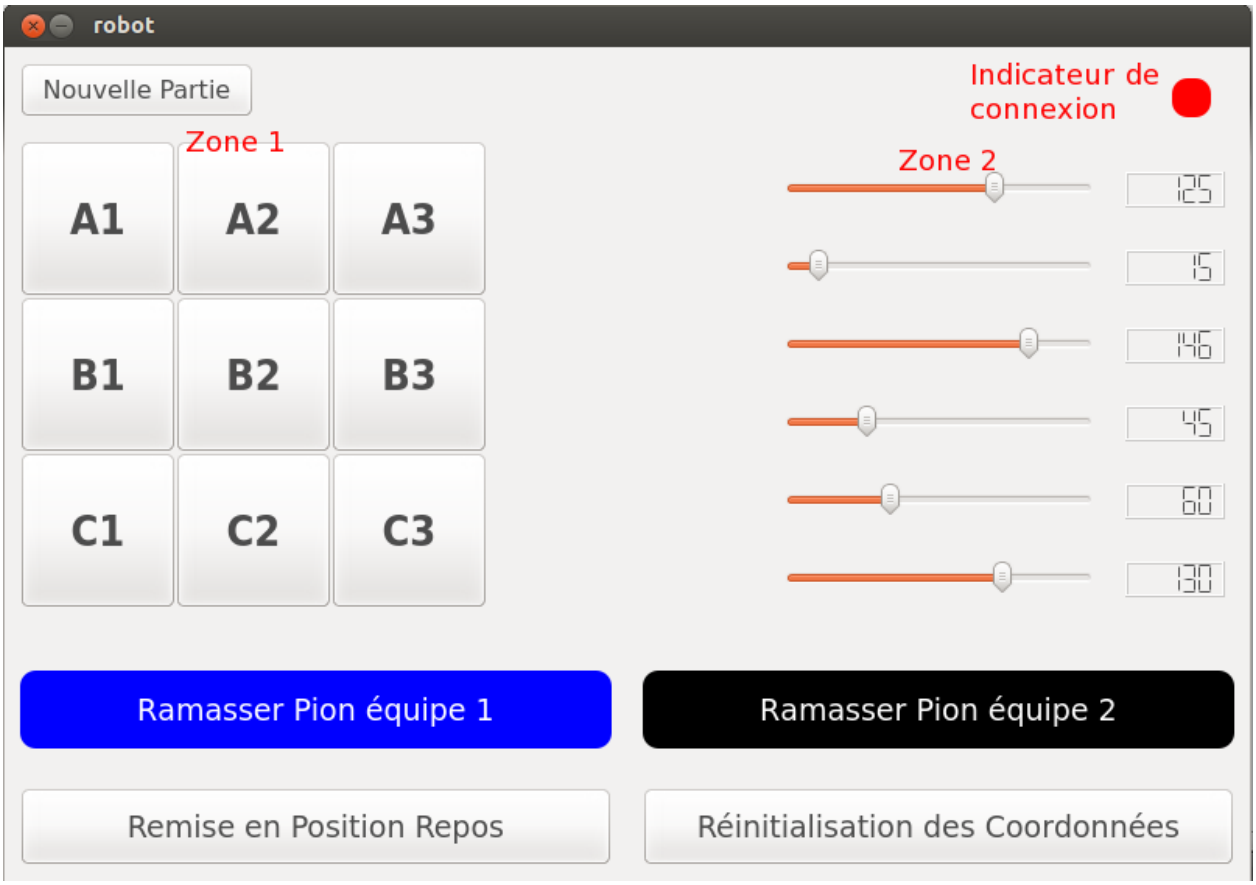


Figure 1 Annexe 2 : capture d'écran de l'interface graphique

La zone 1 correspond à la zone de jeu en tant que telle. L'appui d'un bouton provoque le déplacement du robot à la position référencée pour ce bouton (pour l'instant ces positions ne sont pas modifiables sans toucher au code, mais c'est un axe de développement prévu).
La Zone 2 permet de jouer directement sur la position des 6 servomoteurs du robot. Ainsi un déplacement du premier curseur provoquera le déplacement du servomoteur de la base. Ceci pourra servir à l'avenir pour corriger la position du bras robot, voire pour modifier la grille (avec le bouton réinitialisation des coordonnées, non implémentée pour l'instant).

L'indicateur de connexion en haut à droite permet de savoir si la liaison est opérationnelle (vert), ou non (rouge). Si le voyant est rouge, il peut être judicieux de relancer le programme et de s'assurer que le robot est bien connecté sur le port /dev/ttyACM0 (minicom peut fournir cette information).

Les boutons « Ramasser Pion équipe 1/2 » ordonnent au robot suivant le tour en cours d'aller chercher un pion dans la réserve de l'équipe considérée. A noter que la sélection d'une case après usage d'un de ces boutons la colore en la couleur considérée, permettant un suivi de la partie en temps réel sur l'interface.

- **Pilotage du Robot par Leap Motion**

Prérequis :

- Avoir le Leap Motion branché et le SDK installé
- Avoir Qt installé sur un environnement UNIX
- Avoir Qt Creator
- Le bras robot dûment connecté avec le programme téléversé

- **s’assurer que le robot est bien connecté sur le port /dev/ttyACM0 (seul supporté pour l’instant)**

Une fois le Leap Motion branché, il faut lancer QtCreator et sélectionner le projet dans le dossier /ROBOT_SANS_GUI/robot. Il faut ensuite lancer le programme en pressant CTRL R.

Le Leap Motion est alors actif, il faut commencer par désigner les deux angles extrêmes du damier virtuel pour l’initialiser. Pour cela pincez avec vos doigts dans les airs. Une fois ceci fait, en déplaçant vos doigts comme cela est montré sur la vidéo du projet vous pouvez sélectionner une case où placer le pion. Dans la console s’affiche les informations que remonte le Leap Motion, en particulier la case survolée puis sélectionnée.

Annexe 3 : Manuels du programmeur

1. Les outils indispensables

Minicom

Dans le terminal

```
$ sudo apt-get install minicom
```

puis pour faire les réglages :

```
$ minicom -s
```

Régler le port écouté sur /dev/ttyACM0

Qt Creator

L'IDE incluant l'API Qt est disponible sur le site officiel :
http://qt-project.org/downloads

Pour installer QextSerialPort :
https://code.google.com/p/qextserialport/

Un README explique bien les modalités d'installation. Voilà un rapide résumé des étapes pour une utilisation directe:
- Télécharger la librairie
- La placer dans un dossier de votre projet, par exemple LIBS
- Ajouter à votre fichier de projet :

```
include(LIBS/qextserialport/src/qextserialport.pri)
#include "qextserialport.h" // dans vos .h
```

Leap SDK

Les librairies pour le Leap Motion sont téléchargeables sur le site officiel, selon votre plateforme.
https://developer.Leap Motion.com/downloads
Pour l'installer et l'utiliser sous Linux, cette vidéo montre la marche à suivre :
http://vimeo.com/71036624

2. Réglages en tous genres

- Utilisation de l'interface graphique sous Windows :
Il faut modifier le code fourni. Ouvrez MaFenetre.cpp et en haut du fichier modifiez :

```
m_serialPort = new QextSerialPort("/dev/ttyACM0");
```

en

```
m_serialPort = new QextSerialPort("COM1");
```

où COM1 est le nom du port utilisé.
Nota : La même méthode peut être utilisée pour modifier le nom du port sous UNIX.

- Utilisation du programme avec le Leap Motion sur une machine 64 bits
Avec Qt Creator, ouvrez le projet, remplacez x86 par x64, ça doit donner quelque chose comme ça :

win32:CONFIG(release, debug|release): LIBS += -L$$PWD/lib/x64/release/ -lLeap
else:win32:CONFIG(debug, debug|release): LIBS += -L$$PWD/lib/x64/debug/ -lLeap
else:unix: LIBS += -L$$PWD/lib/x64/ -lLeap

- Utilisation du programme avec le Leap Motion sur Windows
Bien que théoriquement possible, cela n'a pas été mené au bout. Néanmoins les librairies spécifiques à Windows existent, reste à savoir comment dire à Qt de se servir des DLL.

3. Le code

a. Liaison série

L'instanciation de la classe Qextserialport est faite dans le constructeur de MaFenetre pour la partie interface graphique, à l'initialisation du Leap Motion pour la partie sans interface.

On utilise ensuite une simple fonction sendData**N** où N est le numéro du servo concerné pour envoyer les données au robot suivant le formalisme vu dans le rapport. Pour mémoire il faut s'exprimer à lui de la manière suivante :
& N ANGLE
où N est le numéro du servomoteur à piloter et ANGLE la valeur d'angle à lui donner.

b. Partie code avec Interface Graphique :

Pour obtenir cette partie du code : ouvrir Qt puis sélectionner **file>open File or Project** et sélectionner le fichier **robot.pro** contenu dans le dossier **interface_graphique/robot.**

Le code se compose de trois fichiers :
- MaFenetre.h : qui définit le prototype de la classe *MaFenetre*
- MaFenetre.cpp : qui explicite le constructeur de la classe *MaFenetre* ainsi que les slots créés
- main.cpp : qui instancie la classe *MaFenetre* et l'affiche

Une particularité du développement de projet sous Qt est qu'il crée un fichier .pro (ici robot.pro) qui représente une sorte de Makefile permettant de lier les headers aux codes sources et d'inclure les librairies nécessaires au projet.

Dans notre projet, nous avons notamment besoin de la librairie *qextserialport* ainsi que de la configuration qwidget.

MaFenetre.h :

Commençons par le header de notre projet. Il inclut bon nombre de librairies permettant de mener à bien le projet :

```
#include <QApplication> -> créationd'une application
#include <QWidget> -> création de widgets
#include <QPushButton> -> création de boutons poussoirs
#include <QSlider> -> création de sliders
#include <QLCDNumber> -> création d'afficheur LCD
#include <QMessageBox> -> création de message d'erreur ou
   d'information
#include <qextserialport.h> -> permet de gérer la liaison
   série
#include <stdio.h>
```

Ensuite apparait le prototype de la classe *MaFenetre* qui contient des méthodes publiques : son constructeur et son destructeur; des slots publics (toutes les méthodes implémentées pour permettre le déroulement de la partie de morpion) et des attributs privés. Ces attributs sont tous les widgets présents dans la fenêtre, ainsi que la liaison série et les variables *joueur, tour_eq1* et *tour_eq2* permettant le bon déroulement de la partie.

MaFenetre.cpp :

Cette partie du code est la plus compliquée et, comme vous pouvez le voir, la plus longue.

Tout d'abord, on a le **constructeur** de la fenêtre : *MaFenetre().* Dans ce constructeur, d'abord on l'hérite de la classe *QWidget* pour permettre de créer une fenêtre. Ensuite, on commence par instancier la liaison port série. Puis on met en place tous les éléments de la fenêtre : d'abord la fenêtre en elle-même puis ensuite tous les widgets (boutons poussoirs, sliders, …). Enfin, on finit par connecter à l'aide d'un *connect* tous les widgets aux slots dont nous avons créé les prototypes dans le header.

Ensuite viennent les **slots**. Le tout premier est *RemisePosRepos()* qui permet, comme son nom l'indique de remettre le robot en position de repos, mais pour cela, il faut, s'il possède un pion, qu'il le dépose d'abord d'où le *if(...)*.

Après viennent les **slots correspondant à l'action de déposer un pion sur une case** du damier de jeu. Chaque slot est développé de la même façon : d'abord on regarde la valeur de la variable *joueur* qui va nous indiquer quel joueur a pris un pion (équipe 1 ou 2) et si le robot possède bien un pion. La case va alors être coloré avec la couleur du pion sélectionné (bleu pour l'équipe 1 et noir pour l'équipe 2) grâce à la fonction *setStyleSheet(...)* et ne pas être modifiée si aucun pion n'est déposé. Par ailleurs, le code permet de désactiver la case afin qu'aucun autre pion ne puisse être déposé sur cette même case (*setEnable(false)*). Enfin, chacun de ses slots se finira par une remise en position de repos du robot pour plus d'esthétique et pour limiter des pics de courant trop importants.

Après nous avons les **slots de ramassage de pions**. Les deux fonctionnent de manière identique. D'abord on vérifie que le robot n'est pas déjà en possession d'un pion, auquel cas il doit d'abord le déposer avant d'en ramasser un autre. On vérifie également quel est le joueur qui a joué au tour précédent et on affiche un message d'erreur si jamais le même joueur essaye de reprendre un pion à l'aide de *QMessageBox*. Ensuite, on regarde à quel tour on se trouve à l'aide de la variable *tour_eq**n*** pour savoir quel pion aller chercher et on va le prendre. On affichera également un message quand l'équipe n'a plus de pion, ce qui ne doit, normalement, pas se produire en mode de jeu normal.

Ensuite, on a le **slot de réinitialisation de la partie** : *NouvellePartie().* On vérifie que le robot n'a pas de pion en sa possession (avec *joueur*). Si c'est le cas il le dépose dans une partie hors du damier de jeu (pour pas le poser sur une case déjà pleine en cas de partie où le damier est entièrement rempli). Puis ensuite, toutes les cases sont réinitialisées (*setStyleSheet(default)* et *setEnable(true)*) et on réinitialise les variables de jeu (*joueur = 0 ; tour_eq**n** = 1*).

Le **slot de réinitialisation des coordonnées** : *reinit()* n'a pas été implémenté ici. Comme expliqué dans le rapport, nous n'avons pas eu le temps de mettre en place les équations de cinématique inverse, ainsi cette méthode ne fait qu'afficher un message d'information.

Finalement, nous avons les **slots d'envoi des données au robot**. Dans ces slots on crée un tableau de caractère dans lequel on stocke la donnée des angles à transmettre (qui sont donnés par la valeur appliquée aux sliders) + le numéro du servo (donné par le slider modifié) + le caractère spécial qui détermine le début de la trame envoyée au bras de robot (&).

Petite subtilité qu'il faut bien comprendre : Pour contrôler le robot, on passe par l'intermédiaire des **sliders**. En effet, lorsqu'on clique sur une case, le slot en question va modifier la valeur des sliders et c'est cette modification des sliders qui va engendrer un signal qui va appeler les slots d'envoi de données au robot via liaison série.

Remarque : l'envoi des données via liaison série se fait séquentiellement. Si le slider1 est modifié puis le 2, alors le slot *sendData1* va s'effectuer (le robot va alors déplacer selon le servo 1) puis ensuite le slot *sendData2* va provoquer le déplacement du servo 2 etc…

main.cpp :

Dans ce fichier, il n'y a rien de plus à dire si ce n'est que la fenêtre est affichée grâce à la méthode *show()*.

c. Avec le Leap Motion

Le détail du fonctionnement du détail est explicité directement dans le rapport. La connexion avec la liaison série se fait au moyen d'une classe *Serie* contenant l'ensemble des fonctions relatives à la liaison série et au déplacement du robot. Elle est instanciée au sein de la classe Sample Listener qui définit la routine que doit exécuter le Leap Motion.

Annexe 4 : Data sheets

DFRduino Romeo-All in one Controller V1.1(SKU:DFR0004)

Introduction

DFRduino RoMeo V1.1

RoMeo is an All-in-One microcontroller especially designed for robotics application. Benefit from Arduino open source platform, it is supported by thousands of open source codes, and can be easily expanded with most Arduino Shields. The integrated 2 way DC motor driver and wireless socket gives a much easier way to start your robotic project.

Specification

- Atmega 168/328
- 14 Channels Digital I/O
- 6 PWM Channels (Pin11,Pin10,Pin9,Pin6,Pin5,Pin3)
- 8 Channels 10-bit Analog I/O
- USB interface
- Auto sensing/switching power input
- ICSP header for direct program download
- Serial Interface TTL Level
- Support AREF
- Support Male and Female Pin Header
- Integrated sockets for APC220 RF Module and DF-Bluetooth Module
- Five I2C Interface Pin Sets
- Two way Motor Drive with 2A maximum current
- 5 key inputs
- DC Supply：USB Powered or External 7V~12V DC。
- DC Output：5V /3.3V DC and External Power Output
- Dimension：90x80mm

DFRduino RoMeo Pinout

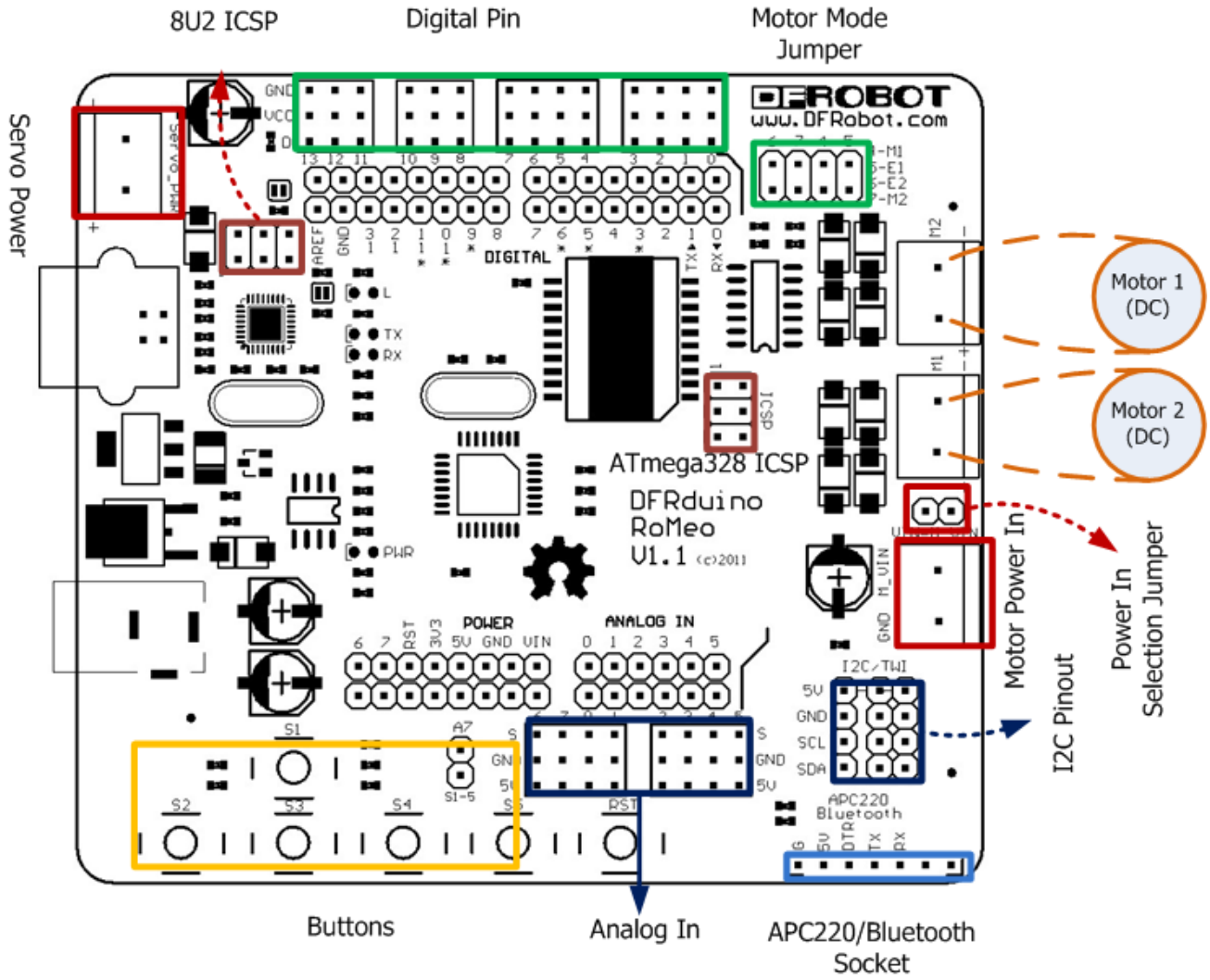


Fig1: Romeo Pin Out

The picture above shows all of the I/O lines and Connectors on the Romeo, which includes:

- One Regulated Motor Power Input Terminal (6v to12v)
- One Unregulated Servo Power Input Terminal (you supply regulated 4v to 7.2v)
- One Servo input power selection jumper
- One Serial Interface Module Header for APC220/Bluetooth Module
- Two DC Motor Terminals – Handles motor current draw up to 2A, each terminal
- One I2C/TWI Port – SDA, SCL, 5V, GND
- One Analog Port with 8 analog inputs – Analog input 7 will be occupied when connecting "A7" jumper
- One General Purpose I/O Port with 13 I/O lines – 4,5,6,7 can be used to control motors
- One Reset Button
- Jumper bank to Enable/Disable Motor Control

Bras robotique à 6 degrés de liberté

Présentation

Le bras robotique à 6 degrés de liberté permet des mouvements rapides, précis et répétables. Il comprend: une base rotative, une épaule mono-plan, un coude, un poignet et une pince fonctionnelle. Il est 100% utilisable avec les cartes DFRduino Duemilanove 328, DFRduino Romeo Tout-en-un et les cartes Arduino Duemilanove standard.

Documentations

- Un article sur le construction de ce robot sur l'excellent site Pobot (que je vous recommande d'ailleurs pour la qualité de son contenu et l'aide qu'il peut vous apporter)
- La vidéo détaillant l'installation est également à votre disposition
- Le guide d'installation se trouve ici.

Spécifications

- Tension: +4.8-7.2 V
- Courant: 2000mA
- longueur du bras: 320mm
- Poids: 600g

Liste des composants

- Pince(x1)
- Support en U (x3)
- Support multifonction (x4)
- Support en L (x1)
- Roulement à bille (x3)
- Base rotative (x1)
- Servo Hitec 311 (x2)
- Servo DF05BB (x1)
- Servo DF15MG (x2)
- Servo HS422 (x1)
- Set de vis (x16)

Annexe 5 : Diagramme de Gantt

	Semaines/ Tâches	**S4**	**S5**	**S6**	**S8**	**S9**	**S11**	**S12**	**S13**	**S14**	**S15**	**S16**	**S17**	**Bilan**
	Diagramme Fonctionnel													
	Diviser le travail													
	Diagramme de Gantt													
	Liste des tâches													
	Montage du Robot													
Partie Communication Bras de Robot	Apprendre à gérer Qt et à coder en C++													
	Code Robot sous Arduino													
	Code liaison série													
	Code interface graphique													
Partie Leap Motion	Prise en main du Leap Motion													
	Compréhension des codes fournis par constructeur													
	Apprendre C++													
	Programme fonctionnel pour le bras de Robot													
Mise en Commun	Mise en commun du code Leap Motion avec Robot													
Rapports	Rédiger le pré rapport													
	Rédiger le rapport final													
Man Power	Astrid	6h	4h	5h	4h	4h	4h	8h	4h	8h	5h	6h	12h	**70h**
	Pierre	6h	4h	5h	4h	4h	4h	8h	4h	8h	5h	6h	12h	**70h**
	Fanny	6h	4h	5h	4h	4h	4h	8h	4h	8h	5h	6h	12h	**70h**
	Olivier	6h	4h	5h	4h	4h	4h	8h	4h	8h	5h	6h	12h	**70h**
	Total	24h	16h	20h	16h	16h	16h	32h	16h	32h	20h	24h	48h	**280h**